Distributed Approach for Peptide Identification

Naga V K Abhinav Vedanbhatla

Western Kentucky University, nagavenkatakrishnaabhinav.vedanbhatla736@topper.wku.edu

Follow this and additional works at: http://digitalcommons.wku.edu/theses

Part of the Analytical Chemistry Commons, Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

http://digitalcommons.wku.edu/theses/1546

This Thesis is brought to you for free and open access by TopSCHOLAR®. It has been accepted for inclusion in Masters Theses & Specialist Projects by an authorized administrator of TopSCHOLAR®. For more information, please contact topscholar@wku.edu.
DISTRIBUTED APPROACH FOR PEPTIDE IDENTIFICATION

A Thesis
Presented to
The Faculty of the Department of Computer Science
Western Kentucky University
Bowling Green, Kentucky

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science

By
Naga Venkata Krishna Abhinav Vedanbhatla
December 2015
DISTRIBUTED APPROACH FOR PEPTIDE IDENTIFICATION

Date Recommended 11/20

Dr. Zhonghang Xia

Dr. Qi Li

Dr. Rong Yang

11/24/15

Dean, Graduate Studies and Research Date
DEDICATION

This thesis is gratefully dedicated to my parents and all my friends who believed in me.
ACKNOWLEDGMENTS

I would like to extend my sincere gratitude to my committee chair, Dr. Zhonghang Xia for providing me this opportunity. Without his guidance and persistent help, this thesis would not have been possible. It has been a great pleasure to work alongside him both as a graduate assistant and as a student.

I would also like to acknowledge Dr. Rong Yang. I am very thankful for the way that she inspired and motivated me. I appreciate Dr. Yang taking the time to review this document. our discussions and her perspective was crucial in completing this research.

I would like to thank Dr. Qi Li who accepted my request to be part of my thesis committee. I am extremely grateful for his assistance and suggestions throught my research and the coursework. I would also like to extend my sincere gratitude to Western Kentucky University for providing me a friendly environment to pursue my masters degree.

I would also like to thank my friends Travis Brummett and Harinivesh Donepudi for helping me in revising this document. I would like to extend my appreciation to my friends Sai Sandeep Jagarlamudi and Kavya Madugula for supporting me during my hard times.

Finally, I would like to extend my gratitude to my parents who has continuously supported me and believed in me each and every moment of my life.
## CONTENTS

DEDICATION ......................................................... iii

ACKNOWLEDGMENTS ................................................ iv

LIST OF TABLES .................................................. viii

LIST OF FIGURES ................................................ ix

ABSTRACT ......................................................... xi

1 INTRODUCTION ..................................................... 1

1.1 Proteins and Peptides ........................................... 2

1.2 Peptide and Protein Identification ............................. 3

1.3 C-Ranker, Peptide Identification and Time ..................... 5

1.4 Research Objectives and Methodology ........................ 7

1.5 Objectives and plan of thesis .................................. 7

1.6 Methodology .................................................... 7

1.7 Advantages ..................................................... 8

1.8 Conclusion ..................................................... 9

2 BACKGROUND ....................................................... 10

2.1 Mass Spectrometry ............................................. 10

2.2 Tandem Mass Spectrometry ..................................... 11

2.3 Post Database Search Methods ................................ 12

2.3.1 PeptideProphet ............................................ 13
LIST OF TABLES

5.1 Input Data Sets for Observations .......................................................... 40
5.2 Hardware Used to Observe Results ....................................................... 41
5.3 Comparison of C-Ranker on distributed approach with C-Ranker on an Apache Hadoop Framework Cluster 1 ......................................................... 49
5.4 Upgraded hardware to compare with cluster2 Hadoop .............................. 49
5.5 Comparison of C-Ranker on distributed approach with C-Ranker on an Apache Hadoop Framework Cluster 2 ......................................................... 49
5.6 Cost Calculation of Apache Hadoop Cluster 1 and Cluster 2 ..................... 50
LIST OF FIGURES

2.1 Distributed System ................................................. 17
2.2 LAN Design ......................................................... 21
2.3 Peer-to-Peer Network ................................................ 22
3.1 Installing Java Compiler Runtime (Initial Screen) ............. 24
3.2 Installing Java Compiler Runtime (License and User Agreement) ... 25
3.3 Installing Java Compiler Runtime (Confirm Installation Settings) .... 25
3.4 Architecture ......................................................... 26
3.5 Data Flow Details in the Original Single-Threaded C-Ranker ...... 28
3.6 Data Flow Details in Distributed C-Ranker (Dividing) .......... 29
3.7 Data Flow For a Single Worker Host ............................. 30
3.8 Data Flow Details in Distributed C-Ranker (Merging) .......... 31
4.1 Download Apache Tomcat ........................................... 33
4.2 Apache Tomcat Extracted in C: Drive ............................ 34
4.3 Apache Tomcat Server Istartupt.batl ............................. 34
4.4 Apache Tomcat Server Running .................................... 35
4.5 ROOT.war file in the Apache Tomcat Directory ................. 36
4.6 ServerRunner.jar File Moved to Home Directory .............. 36
4.7 Cranker Properties File .............................. 37
4.8 ServerRunner.jar Running and Asking for A Port Number ....... 38
4.9 Web Based Interface for Submitting Input File ........................................ 39
5.1 Results for testData.xls ................................................................. 41
5.2 Results for Pbmc_orbit_mips.xls ....................................................... 41
5.3 Results for Pbmc_orbit_nomips.xls ...................................................... 42
5.4 Results for Pbmc_velos_mips.xls ......................................................... 42
5.5 Results for Pbmc_velos_nomips.xls ...................................................... 43
5.6 Memory Usage ............................................................................. 44
5.7 Memory Usage ............................................................................. 44
5.8 Memory Usage ............................................................................. 45
5.9 Memory Usage ............................................................................. 45
5.10 Memory Usage ............................................................................ 46
5.11 Memory Usage ............................................................................ 46
5.12 Memory Usage ............................................................................ 47
5.13 Difference in Memory Usage ............................................................. 48
A crucial step in protein identification is peptide identification. The Peptide Spectrum Match (PSM) information set is enormous. Hence, it is a time-consuming procedure to work on a single machine. PSMs are situated by a cross connection, a factual score, or a probability that the match between the trial and speculative is right and original. This procedure takes quite a while to execute. So, there is demand for enhancement of the performance to handle extensive peptide information sets. Development of appropriate distributed frameworks are expected to lessen the processing time.

The designed framework uses a peptide handling algorithm named C-Ranker, which takes peptide data as an input then identifies the accurate PSMs. The framework has two steps: Execute the C-Ranker algorithm on servers specified by the user and compare the correct PSM's data generated via the distributed approach with the normal execution approach of C-Ranker.

The objective of this framework is to process expansive peptide datasets utilizing a distributive approach. The nature of the solution calls for parallel execution and hence a decision to implement the same in Java has been taken. The results clearly show that distributed C-Ranker executes in less time as compared to the conventional centralized C-Ranker application. Around 66.67% of the overall reduction in execution time is shown with this approach. Besides, there is a reduction in the average memory usage with the distributed system running C-Ranker on multiple servers.

A great significant benefit that may get overlooked is the fact the distributed C-Ranker can be used to solve extraordinarily large problems without incurring expenses for
a powerful computer or a super computer. Comparison of this approach with An Apache Hadoop Framework for peptide identification with respect to the cost, execution times and flexibility were discussed.
Chapter 1

INTRODUCTION

Bioinformatics is the field of study of biological data with the aid of computational methods. The field deals with a blend of Computer Science, Statistics and Biology. Machine learning, which can be thought of as Statistics in disguise, is used in a large number of bioinformatics applications. The experience gained by the application of Machine Learning in a variety of fields has also contributed to its applications in bioinformatics [Bhaskar, Hoyle, and Singh, 2006]. Moreover, it has aided the advances in information technology and the development of mature tools to analyze data and build models for inference, e.g. weka [Frank, Hall, Trigg, Holmes, and Witten, 2004].

The study of human gene sequence can reveal information about diseases and drug delivery development techniques [Collins, 1999]. With the amount of genomic information available exploding profusely, it is becoming necessary to use computers for procuring, storing and analyzing biological information. Moreover, bioinformatics deals with the simulation of DNA structures and molecular interactions. Text mining helps us deal with large amounts of biological information available. With the ever increasing amount of data stored and generated, text mining becomes more and more crucial [Zou, Lin, Liu, and Guo, 2011]. Machine learning provides tools and methods for learning patterns from data and making reasonable predictions over them [Tan et al., 1999].

A common application in bioinformatics is the identification of peptide/proteins.
Protein identification is important for drug development and quality control of drugs [De-
meule, Regina, Che, Poirier, Nguyen, Gabathuler, Castaigne, and Beliveau, 2008]. The
method that we employ in this thesis is based on peptide mass fingerprinting, where a pro-
tein is identified by comparing the masses of the peptides obtained after its decomposition
with trypsin and a probability based technique is used to carry out protein identification
[Cottrell and London, 1999].

1.1 Proteins and Peptides

Proteins are macro-molecules. The problem of finding the structure of proteins
have been attacked from various angles in the past decades [Pauling, Corey, and Branson,
1951]. They consist of long chains of amino acids connected to each other with bonds
called peptide bonds.

Smaller chains of amino acids are called peptides. A protein is characterized by
the sequence of amino acids as they occur in the protein. Proteins are involved in almost
every biological process happening in an organism’s body including metabolic activities,
transfer of molecules from one part to another, replication of the DNA. It is because of this
prevalence of proteins in biological processes that protein identification forms an important
part of drug development to target specific metabolic pathways.

The sequence of amino acids in a protein determines the three-dimensional structure
that would result from its folding, known as native conformation. There are mainly four
aspects of protein structure: primary, secondary, tertiary and quaternary.

Primary structure of a protein is determined by the sequence of amino acids con-
tained in the protein. Secondary structure is the structure repeating sub-parts of a protein.
Tertiary structure talks about the global shape of the protein molecule which is determined
by the spatial interactions of the sub-parts that make up the secondary structure. Quaternary structure comes into discussion when we have more than one protein molecule interacting with each other and forming a specific shape. All these structures refer to a domain that varies the region we focus into while talking about structure of the protein molecules.

During the lifetime of a protein, it may evolve by changing its structure. Such changes in the structure are termed conformational changes [Lumry and Eyring, 1954].

Peptides are chains of amino acids, with the number of amino acid residues being in the range of 10-50. Peptides with more than 200 amino acid residues are called proteins. Peptides are the building blocks of proteins. Polypeptide bonds join multiple peptide strings together and make a protein [Doonan, 2002].

In bottom-up proteomics [Zhou, Lu, Wang, Borhan, and Reid, 2010], a protein is first digested into peptides and then each peptide is identified individually to infer the protein identity. The larger the sequence of amino acid that is to be determined, more the computational resources that would be required. So, peptides are usually identified by most database search methods rather than whole proteins. The digestion of the protein sample is carried out multiple times to obtain the mass-spectra of constituent peptide. The overlap that is obtained between multiple runs of the process helps us identify the origin protein sample [Aebersold and Mann, 2003].

1.2 Peptide and Protein Identification

Identification of peptides/proteins starts with employing the unknown sample with an enzyme like trypsin. Trypsin breaks the protein into its constituent peptides. These peptides are then ionized to form their respective ions. The ions pertaining to a specific peptide
are then sent to a mass spectrometer for its identification. Spectrometry has contributed a lot to advances in our knowledge of protein and peptide structure, as well [Biemann, 1988].

To identify an unknown peptide, a common tool used is SEQUEST [MacCoss, Wu, and Yates, 2002]. SEQUEST has a database of proteins and can compute the peptides that would result from its decomposition. Moreover, it is capable of computing the theoretical mass spectrum of individual peptides. The observed mass spectrum of the unknown peptide is compared with the theoretically derived mass spectra of all the potential peptides in the SEQUEST database. The cross correlation [Eng, Fischer, Grossmann, and MacCoss, 2008] between the theoretical spectrum and the observed mass spectrum is used as a score to rank potential peptides. The peptide with the highest score is selected as the output. In order to reduce the search space, SEQUEST also stores the intact masses of peptide ions which are compared with the observed mass of the unknown peptide. SEQUEST can return a collection of peptides.

SEQUEST produces a list of matches for the unknown peptide [Käll, Storey, MacCoss, and Noble, 2007]. Each of these matches is called a Peptide-Spectrum Match (PSM). So, a set of PSM records is sent as input to a new tool called C-Ranker which then figures out which of the records are more reliable than the others. C-Ranker computes a score for each PSM record which gives us an indication of how reliable it is. The computation of the score for a PSM record can be done in parallel with other records. When the number of records is very large, it becomes necessary to introduce parallelism in the whole process.

Peptide identification is crucial in the identification of proteins. A protein consists of a peptides in a specific proportion in terms of their masses. This can be thought of a fingerprint of the protein. To identify a protein, we figure out the constituent peptides and
their respective masses. Then, a database of known proteins is searched and the one that
matches the observed ratios of peptides most closely is figured out.

### 1.3 C-Ranker, Peptide Identification and Time

C-Ranker is a post-database tool, i.e. it processes the output of a database search
engine like SEQUEST. The output of a classifier is binary wherein a +1 indicates a target
PSM and a -1 indicates a decoy PSM. Of all the PSMs marked as target PSMs, almost half
of them are false. So, figuring out quality PSMs remains a challenge. This challenge is
tackled by C-Ranker. C-Ranker takes an SVM-based clustering approach to classification.
The idea is that target PSMs are similar to each other in terms of their attributes and hence
cluster in the attribute space [Liang, Xia, Niu, Link, Pang, Wu, and Zhang, 2013].

FC-Ranker scoring methods is a fuzzy scoring method, meaning that it assigns to
each identified target PSM a probability, i.e. the probability that the target PSM is really a
target PSM and not a decoy one. The scores are obtained by a combination of a discriminant
function and a silhouette index [Liang et al., 2013]. Silhouette value is in clustering to see
how strongly a specific data point belongs to its assigned cluster. The values of the two are
iteratively modified until the algorithm halts. It has been observed that the scores assigned
to each PSM are representative of the likelihood of a PSM being a target PSM.

A need for a fuzzy method is evident from the fact that most of the PSMs output by
database searching do not have reliable positives. This non-determinism in the +1 labels
in the training dataset distorts the model that a learning algorithm would see. Hence, a
non-fuzzy method is likely to perform poorly compared to a fuzzy one. For very large
scale problems the number of input records is huge and hence a serial computation of the
peptides is not a good idea. Hence, a parallel setup is needed and implemented. So, our implementation can run C-Ranker on n number of servers simultaneously.

Each of these servers would then generate a score for the PSM records it was handed with. Then these scores can be combined to compute the most probable candidates, i.e the ones that were scored the highest of all. This technique is similar to Map-Reduce wherein a computation is broken down into chunks which can be acted upon at the same time. Each of these chunks results into their respective results. This is the Map part of the computation. Now, we need to reduce or fold the results obtained from individual chunks to summarize the computation.

A common trade-off that one is faced with while doing parallel computing is that of figuring out the right granularity. We need to decide the right size of individual sub-problems handed to each processor. Keeping the granularity very high would decrease the CPU utilization at the end of computations and some of the processors may just not get any work to do. Keeping the granularity very small, we risk the possibility of making the coordination overhead larger than the gains achieved with parallelism. More time is spent making the sub-processes coordinate with each other while each of the sub-processes actually does a very small computation. It’s an open problem to automate parallelism [Blume, Eigenmann, Hoeflinger, Padua, Petersen, Rauchwerger, and Tu, 1994]. Most programming that deals with parallel computation involves annotations for specifying the degree of parallelism to add. Similarly, the application that we developed takes the number of nodes in the network among which the task is divided and distributed as an input [Marlow, 2013].
1.4 Research Objectives and Methodology

C-Ranker was used on a single machine. When the size of the PSM records increased, execution time becomes impractical or inconvenient. For example, for a dataset having 400000 PSM records, it may cost about 5 hours on a PC with CPU Intel i5-2400 2.10 GHz and memory 8GB. So, the main goal of this thesis is to distribute this computation among the nodes in a network without compromising its correctness and without making any significant changes in the original C-Ranker algorithm.

1.5 Objectives and plan of thesis

• Create a framework to execute C-Ranker on distributed node.

• Design such that it may work with other post database searching algorithms like C-Ranker with minimal changes

• Compare the time taken of generated distributed output of C-Ranker with actual output

• Make sure C-Ranker algorithm is well executed on the set of predefined nodes.

The obvious step is to understand the C-Ranker algorithm, observing how the algorithm can be parallelized. It is observed that the records can be parallel processed. So, a framework for the division and distribution of PSM records to different computing nodes and then the accumulation of the computed results to a centralized node is built.

1.6 Methodology

Dataflow Definition: Defining how data, i.e. PSM records are to be divided among the nodes which are available as individual computing units. The records are equally di-
vided among the nodes that the user enters as input to the interface we make available to them.

**Architecture Design** : The architecture is mainly a server-client architecture rather than a peer-to-peer architecture. There is a centralized node, called the client and there are distributed computing nodes called servers. The client’s job is to divide the records into pieces that can be handed to individual nodes for computation. The client sends this data to the nodes through the network. After the nodes have completed their respective jobs they send back the results to the client. The client also summarizes the results of all the computing nodes.

**Architecture Implementation** : This step involves the implementation of the drivers at the client side. Divider and Merger are the drivers implemented at the client side. The divider driver divides the input into the number of servers used by the user. The merger driver combines the outputs from the servers into a single output.

**User Interface Design** : A user-interface for taking the records and showing the results in form of charts and tables so that the user can compare the results is built in this step. A web based user interface was created.

### 1.7 Advantages

C-Ranker acts on the records in a sequential manner while the computational task can be parallelized and hence we can take advantage of multiple processors. The gains obtained are more important because cheap commodity hardware when replicated costs less than a single very high-end system. That’s precisely the reason most data-centers use
commodity CPUs connected through a computer network. So, implementing C-Ranker in a distributed setting is important in terms of performance and costs trade-offs involved.

Creating a distributed framework for C-Ranker implies that we can solve even larger problems now. The ability to solve large problems open up interesting research areas in drug development and delivery where peptide/protein identification plays a crucial role.

1.8 Conclusion

We started out with the aim of implementing the C-Ranker algorithm in a distributed setting without making changes to the original steps of execution. We obtained that the executing time can be reduced by a factor equal to the number of computational nodes employed for the task.
Chapter 2

BACKGROUND

2.1 Mass Spectrometry

Mass Spectrometry is a technique used for the analysis of chemical compounds to find out their structure and composition [Aebersold and Mann, 2003]. It mainly consists of finding the mass-to-charge fingerprint or a mass-spectrum so to speak of the sample under consideration. The chemical sample to be analyzed is first ionized by bombarding it with the accelerated electrons. This causes the compound to break into charged fragments. These charged fragments are then made to accelerate using an electric field. There is an electric or magnetic field applied in the direction perpendicular to the accelerating field, called the transverse field. This field is responsible for bringing about a deflection in the charged particles. The amount of deflection caused is dependent on the mass-to-charge ratio. The charged fragments with higher mass-to-charge ratios are deflected less as compared to the ones with smaller mass-to-charge ratios. These moving fragments are then received on a screen consisting of electron multipliers. The amount of fragments amount at a specific spot on the target screen, gives an indication of the amount of that particular ion in the chemical sample. The whole data, consisting of the fragments and their response on the screen, gives us a fingerprint of the chemical sample. This fingerprint can then be compared with the fingerprints/mass-spectra of known compounds and hence the composition of the chemical sample is figured out.
A device that carries out mass spectrometry is called a mass spectrometer. Here, we discuss the structure and design of such a device [Wiley and McLaren, 1955]. A mass spectrometer mainly consists of three parts:

**Ion Source**- Converts the sample into constituent ions and their isotopes. Initially, the sample is vaporized and then ionized by using various techniques.

**Mass Analyzer**- Sorts the generated ions according to their mass-to-charge ratio with the help of applied electric and magnetic fields.

**Detector**- Collects ions and depending upon the amount arriving at a particular sensing point, outputs a value indicative of the abundance of the ion in the sample. A type of electron multiplier is used as the detector. It senses the passing of an ion or an ion hitting its surface and generates a current in response. This current is then considerably amplified. The detector along with the current gives us a point in the mass-spectra of the sample. The spatial position of the detector on which the measurement is done gives us the mass-to-charge ratio, i.e. the identity of the ion. The current produced by the detector gives us an indication of the amount of the ion present in the chemical sample.

### 2.2 Tandem Mass Spectrometry

Tandem Mass Spectrometry consists of multiple stages of mass spectrometry. There is fragmentation between two stages of mass spectrometry. To do multiple mass spectrometry multiple times in a sequence, a temporal or spatial technique may be employed. In Spatial Tandem Mass Spectrometry, there are multiple mass spectrometers physically separated...
from each other and the sample constituents travel these mass spectrometers in sequence. On the other hand, in a temporal setting, there is only one spectrometer and the different stages of mass spectrometry are separated in time rather than in space [McLafferty, 1981].

Tandem Mass Spectrometry is very important in protein sequencing. Statistical models have been proposed for identifying proteins with the help of tandem mass spectrometry [Nesvizhskii, Keller, Kolker, and Aebersold, 2003]. Usually, proteins are decomposed into peptides before their tandem mass spectrometry is carried out. The decomposition is carried out by an enzyme like trypsin. Then mass-spectrometry eventually adds a peptide-tag to each peptide that the protein decomposes into [Hunt, Yates, Shabanowitz, Winston, and Hauer, 1986]. This process may be carried out multiple times. Tandem Mass Spectrometry has recently gained more popularity because of its advancement in computing technologies and our ability to deal with ever larger databases of protein sequences [Domon and Aebersold, 2006].

2.3 Post Database Search Methods

PeptideProphet, Percolator, and C-Ranker are all post database search tools in peptide identification. They use slightly different machine learning algorithms to classify PSMs into target and decoy PSMs. The following paragraphs will discuss each of them one by one. The usual database search tool that is preceded by these methods is SEQUEST. The output PSMs generated by SEQUEST aren’t reliable. There is a larger fraction of false positives in the results produced by SEQUEST. We discuss how each of these tools validate the assignments made by SEQUEST and see why C-Ranker is our tool of choice for this particular task. We then talk about our choice for the platform used for implementing a distributed version of the C-Ranker tool.
2.3.1 PeptideProphet

PeptideProphet is used for validating the peptide assignments made by database search programs, e.g. SEQUEST. It learns the distribution of the scores and properties of the peptides for correct and incorrect matches. These features, as mentioned in the tool’s website, include the number of termini compatible with enzymatic cleavage (for unconstrained searches), the number of missed enzyme cleavages, the mass difference with respect to the precursor ion, the presence of light or heavy cysteine (for ICAT experiments), and the presence of an N-glycosylation motif (for N-glycosylation capture experiments).

PeptideProphet employs the expectation maximization algorithm [Moon, 1996] to build a statistical model used for distinguishing correct from incorrect database search results, along with probabilities of them being correct. PeptideProphet is able to filter large volumes of tandem (MS/MS) spectra results with predictable error rates.

The statistical model used by PeptideProphet is based on Bayes Theorem [Ma, Vitek, and Nesvizhskii, 2012]. Usually, database search programs output a search score or a set of search scores for each search result. Suppose the search scores are labelled $x_1, x_2, x_3, x_4, x_5$. Let’s denote this set of scores with $S$. Then the probability that a result is correct given that it has $S$ as it set of scores is given by [Keller, Nesvizhskii, Kolker, and Aebersold, 2002]:

$$P[Correct|S] = \frac{P[Correct]P[S|Correct]}{P[S]}$$

As the number of scores used by the database search program increases, the calculation of joint probabilities for the equation becomes more difficult. A discriminant function
analysis is adopted by PeptideProphet in order to cope with this problem. From the given set of search scores, it obtains a single score which is obtained as a linear combination of the search scores contained in the set $S$. A discriminant function may be defined as follows [Fryback, 1978]:

$$F(x_1, x_2, x_3, x_4, x_5) = C_0 + \sum_{i=1}^{n} C_i x_i$$

Now, this value $F$ can be used in place of $S$ in the Bayes equation to compute the probabilities.

2.3.2 Percolator

Percolator uses a semi-supervised machine learning algorithm for validating the peptide assignments made by database search programs [Käll, Canterbury, Weston, Noble, and MacCoss, 2007]. The search scores returned by SEQUEST or a similar program can be considered reliable for very high values of the scores and for very low values. It is in the intermediate values that we are unsure about the assignments and hence we are supposed to observe most of the false positives and false negatives in that range of score values [Spivak, Weston, Bottou, Kall, and Noble, 2009].

Percolator learns from the search scores lying at the extremities to train a model which is then used to classify the peptide assignments that we are unsure about. So, the peptide assignments which have very high values of scores are input as the positive examples and the ones with very low values are input as negatives to the supervised learning algorithm. Support Vector Machines are used for carrying out the classification.

Support Vector Machines during the training phase try to find a hyper-plane that
separates the positive training examples from the negative ones in the feature space of the training data. The hyper-plane chosen as the separating plane is the one that maximizes the margin, because such a plane would be the natural choice for the classification boundary. It is obvious that not all data would be linearly separable in the feature space. That is why Support Vector Machines uses what is called the kernel trick [Scholkopf, 2001]. A kernel is a function used to transform data from the feature space to a higher dimensional space, with the hope that in such a higher dimensional space, the data would become linearly separable. The most commonly used kernel for SVM is the RBF kernel.

The default set of features used by Percolator, as has been mentioned on its website, are `PercolatorFeatures mScore, lgDScore, mrCalc, charge, dM, dMppm, absDM, absDMppm, isoDM, isoDMppm, mc, varmods, totInt, intMatchedTot, relIntMatchedTot`

2.3.3 C-Ranker

C-Ranker, on the other hand, uses fuzzy classification methods. It uses fuzzy SVM and incremental fuzzy silhouette index. A fuzzy method is effectively able to reduce the distortions introduced by outliers [Jiang, Yi, and Lv, 2006].

On every iteration of the algorithm, the silhouette index is updated in such a manner that the index indicates the reliability of the PSM when the algorithm terminates. As the target PSMs resemble each other, it makes sense to employ clustering with classification to determine the target PSMs. So, correctly assigned PSMs would form one cluster and the rest of the PSMs would form another with a vaguely changing boundary between them defined by the silhouette index assigned to a PSM at the end of the algorithm.

Experiments have shown that C-Ranker outperforms other post search algorithms [Liang et al., 2013]. In the experiments, an RBF kernel was used for the fuzzy SVM. The
features of the peptides used were– x-correlation, delta-cn, ions, sprank and calcneutral-pep-mass and few others. FC-Ranker outputs more target PSMs than PeptideProphet and Percolator does while they share a large number of PSMs in common [Liang et al., 2013].

C-Ranker was chosen because it outperforms the other two methods that we have discussed above. Moreover, it is easy to parallelize C-Ranker to make it work on a network of computers rather than on a single computer to work on a larger scale. C-Ranker comes with a manual that is easy to follow as to how the application is to be used. Moreover, there is a research team close in our proximity whose research is focused on the same algorithms and hence they could be a great source of information and motivation.

### 2.4 Distributed Systems

A distributed system is a collection of autonomous systems coordinating with each other by some software means to achieve a specific task. The computers that form a distributed system are linked to each other with a computer network [Tanenbaum and Van Steen, 2007].

Centralized systems, as opposed to distributed systems, have a single point of control and failure. There is only one autonomous component with several non-autonomous components controlled by the autonomous entity. Distributed systems, on the other hand, have multiple components operating independently that can fail. A distributed system should be able to overcome the failure of its components [Coulouris, Dollimore, and Kindberg, 2005]. Figure 2.1 shows an example of a distributed system. Each computer of the distributed system has its own memory and runs its own operating system. All the computers are connected through a network.

The key features of distributed system are as follows:
Resource Sharing - The components of a distributed system can share resources located anywhere in the system. There is a component called Resource Manager that handles control of the sharing of resources. Resource Manager provides naming scheme and concurrency. Naming scheme is important for resources to be addressable. A resource sharing model describes how resources are to be shared and how they are to be used [Wang and Morris, 1985].

Openness - Openness of a system refers to the ability to use and provide services from and to the components of the system irrespective of the heterogeneity of the underlying hardware/environment. Implementing openness requires the establishment of well-defined interfaces for the interaction of components. The underlying environment, which is comprised of the hardware, the platform and the languages, should not affect the functioning of the networking. This means that the details such the byte-order should be made irrelevant to the act of communication between components of the system [Tanenbaum and Van Steen, 2007].

Concurrency - Components in a distributed system access and update shared resources in
a concurrent fashion. Integrity of data becomes important in a concurrent setting and may be violated if concurrent updates are not coordinated. Concurrency control becomes crucial in any application that has multiple actors changing some shared state [Bernstein and Goodman, 1981].

**Scalability**- A distributed system’s capabilities can be extended by increasing the number of autonomous systems in the system. In systems that facilitate the easy addition of nodes for scalability, the problem is the communication overhead and gains obtained by scaling. We need to find a trade-off between the two.

**Fault Tolerance**- A distributed system, unlike a centralized system, has multiple points of failure and hence it should continue to work amid failures at these points. The individual autonomous components may fail and be replaced but the system as a whole should continue to work [Jalote, 1994].

**Transparency**- A distributed system should appear as a single coherent entity to its users. It should be able to hide the complexity that it deals with internally. For example, while sending email, we are not aware of how the servers that manage mail throughout the world synchronize or coordinate to provide the content of user’s inbox. The distributed system should be able to hide where a specific shared resource is located, hide failures of specific components, etc. A good distributed system must not expose its complexity to its clients [Mantena, Mantena].

2.4.1 Centralized system Vs Distributed System

A centralized system has a single point of control and failure. It is hard to expand in terms of memory and computational resources. It has a single global clock and a central-
ized system with high-end CPU and large memory is very expensive. There are multiple autonomous units in a distributed system that coordinate with each other to provide a service to its users or accomplish a task. A distributed system can be scaled as per the needs by introducing new computers in the network. It doesn’t have a global clock. Each computer in the system has its own clock. An equivalent distributed system offering the same computational capabilities is cheaper than the centralized system.

2.5 Why not distributed frameworks like Hadoop?

Due to the distributed nature of open-source development, assurance of quality is difficult. In the end, only some needs will be met. It is the solution of unpredictable usability and quality. The programming model is difficult to leverage for more than simple transformational logic, and is also very restrictive. HDFS of Hadoop has no optimizer, so developers need to be sure to optimize their data themselves. As it is dependent on a file system, there is no transaction consistency. This means that the result which we expect from a Hadoop cluster may not be 100 percent accurate. The major drawback with Hadoop is it doesn’t support relational data. If there is any row in a data dependent on any other row, Hadoop is not the right choice. This is the point we’ve decided not to use it and look for a feasible solution in Java.

2.6 Distributed framework effects on C-Ranker

C-Ranker in a distributed system can handle bigger datasets than it would be able to in a centralized setting. It requires less memory per computer and hence each computer can have commodity hardware. It is cheaper to have multiple commodity hardware computers than having a single high-performance high-end system capable of achieving similar goals.

Moreover, as the size of the data that we are dealing with increases, failures become
more and more costly because a failure would mean re-starting the whole computation which may consume a lot of time. So, having a distributed system enables us to keep moving forward in the face of failures.

2.7 Java for Distributed Systems

Java is popular for building distributed applications. There are a couple of reasons that contribute to this popularity [Boger, 2001]. One of the reasons is the availability of seasoned libraries that work on the JVM. The main points that make Java a perfect choice for building distributed systems are as follows:

- Java has support for networking in its core libraries itself, e.g. support for TCP/IP/HTTP/WebSockets. It even has support for security features like SSL in the core libraries of the language.

- Java, from its inception, supports multi-threaded programming with thread creation, locking primitives and multi-threaded data-structures. In a distributed system, multi-threading is important for achieving a constant throughput.

- Java has a few well-thought APIs which can be implemented for custom classes to keep the intuitive interface that most of Java programmers are familiar with.

- Java has been supporting serializing objects into bytes and de-serializing them into objects from very early on. This makes data transfers in a networked setting very handy. Moreover, since Java programs run on the JVM, they can work across multiple architectures and operating systems. This portability adds more power and flexibility making Java a perfect choice for building a distributed system.
Moreover, Java has a huge community of industrial users and that makes finding help on a specific issue that we might encounter easier than for any other platform. JVM is a mature runtime-environment and is highly optimized.

2.8 Distributed Systems in networking point-of-view

LANs are used in data centers to connect hundreds or even thousands of servers. We roughly distinguish two communication levels: First, servers are grouped on racks, linked by a high-speed cable. A typical rack contains a few dozens of servers. Following, a data center consists of one to a large number of racks connected by routers (or switches) that transfer non-local messages. In all cases, servers only communicate through message passing. They don’t share storage or computing resources.

A Peer-to-Peer system is a specific sort of system, where a chart structure assemble
more than a local physical system [Rodrigues and Druschel, 2010]. The physical system we consider here is the Internet. Hubs or associates speak with messages sent over the Internet. The course that unites two companions on the Internet is ordinarily muddle. By abstracting this many-sided quality, a P2P system envisions an immediate connection in the middle of A and B, as though they were specifically associated, when they know the IP locations of one another. A very famous peer-to-peer protocol is the bit-torrent protocol, also has applications that go by that name [Cohen, 2008]. In peer-to-peer systems, the ownership of the file is vague as the file exists within the network, chunks of the file may be present in different nodes while no node having the complete copy of the file. This can be and has been used to breach the copyright laws [Karunaratne, 2012].

![Peer-to-Peer Network](image)

Figure 2.3: Peer-to-Peer Network
Chapter 3

ENVIRONMENT SETUP AND DESIGN

3.1 Overview

This chapter deals with the installation of the platform on which the MATLAB application runs. It talks about the architecture of the distributed C-Ranker that is implemented in this project. We also spend some time talking about the data flow in the system as it crunches results. In the end, we talk about the methods in the source code that are assigned specific tasks in the architecture and data flow.

3.2 Java Runtime Environment

Java programs are compiled with the Java compiler to a portable binary format called Java bytecode. Java bytecode is then interpreted by the Java Virtual Machine. Java Virtual Machine is an abstract computer defined by a specification. This unique combination of compilation and interpretation makes Java programs safe, secure, and portable. Type-safety is added because of the fact that Java is a typed-language. Java programs run in a virtual machine and hence illegal memory access can be detected and avoided. Since Java bytecode only requires an implementation of the Java Virtual Machine on a platform, Java programs run seamlessly on all platforms that have an implementation for JVM.

Java Runtime Environment (JRE) is the name under which Oracle ships its implementation of the Java Virtual Machine bundled with Java Class Library [Radhakrishnan, Vijaykrishnan, John, Sivasubramaniam, Rubio, and Sabarinathan, 2001].
3.2.1 Installing Java Runtime Environment

The first step in setting up the environment running the C-Ranker application, is to have JRE installed.

Figure 3.1: Installing Java Compiler Runtime (Initial Screen)

Figure 3.1 shows the first screen after executing MCRInstaller.exe. We click next to proceed and do the installation. The next screen shows the license and user agreement.

Figure 3.2 shows a screenshot showing the window for the license and user agreement. We accept and proceed.

The next screen asks for a confirmation for the installation settings. The settings include the path into which the runtime would be installed. The default path is C:\Program Files (x86)\MATLAB\MATLAB Compiler RunTime \runtime\win32.
Figure 3.2: Installing Java Compiler Runtime (License and User Agreement) on Windows. After clicking Install, the runtime would be installed automatically. Figure 3.3 shows the final screenshot in the installation process.

Figure 3.3: Installing Java Compiler Runtime (Confirm Installation Settings)
3.3 Architecture

In this section, we discuss the architecture of the distributed framework we set up for the execution of C-Ranker. Fig 3.4 shows the architecture.

The client in Figure 3.4 is the browser wherein a user inputs the input file and is able to see the results. The client browser talks to an Apache Tomcat Server. It is the responsibility of the client to divide the work among instances and send them off to worker computers listening for commands.

The web-server obtains the input file. The input file contains a number of PSM records. The input file is split into multiple files each of which is sent to a separate machine listening on a specific port. We obtain the PSM records in an excel file. The excel file is converted to a text file by the tomcat server. These text files are then divided into several text files each with a header that is exactly the same as the original text file but with the records
divided among them. The records contained in these text files form a mutually exclusive and exhaustive set of the original records that we started with. The several running instances of the C-Ranker application are then handed off these files as excel files while the tomcat server waits for the responses from these instances. It then obtains the split results excel files from individual instances of the C-Ranker application. These files are then merged into a single file and sent to the client browser.

The division of the original input excel file into parts is controlled by a parameter which can be easily configured. This makes this system scalable to more number of instances and hence we can use the same system for larger and larger datasets by increasing the number of worker hosts among which the work would be split.

3.4 Data Flow

In this section, we would talk about how data flows in this framework. A general scheme for making sense of the architecture of a software system is understanding how it manipulates data as time passes. The time here is usually the CPU-time, but from a philosophical point of view, it is the same.

3.4.1 Data Flow for Single C-Ranker

In the single-threaded settings, the data-flow doesn’t have more than one player. The input file, shown as testData.xls in Figure3.5, is first read in the Matlab’s (.m) format with the help of the cranker_read that comes with C-Ranker package. The output of cranker_read is the file thetestData.m, which contains the same PSM records as the original xls file but just in a different format. Now, this file is either given as input to cranker_write which outputs the reliable PSM or is sent as input to cranker_solve
which first scores each PSM and outputs a file with the PSM scored in accord with their reliability.

The output file generated by \texttt{craker\_write} is a text file containing the reliable PSM records, with its name in the format \texttt{result dd-mm-yyyy.txt}.

### 3.4.2 Data Flow for Distributed C-Ranker

Distributed C-Ranker requires more than one hosts to run multiple instances of C-Ranker commands in parallel. Figure 3.6 shows the flow of data in such a setting. To implement the divide-and-conquer approach, and that too in parallel, we must have a divider methods whose \textit{raison d’etre} is to distribute the work into multiple worker hosts. The divider method is represented with a divider gear in the figure because it is the main driving unit of the whole system. It is the point where \textit{distribution} happens in the whole \textit{distributed} system.

The file input to \texttt{divider} is \texttt{testData.xls}, splits into multiple \texttt{split\{i\}\_test.xls} where \textit{i} goes from 1 through the number of servers we chose a
command-line parameter while running the C-Ranker application on the Tomcat Server. Each of these xls files is handed to a separate server where the various cranker methods are executed on it in the right sequence.

Figure 3.7 shows the data flow for each instance of the worker hosts. This is exactly the same as the single-threaded data flow but with the files containing as subset of the original records than the whole of input records.

Now, once each of the workers has computed the results on the sample of records it was handed with, we need some mechanism to merge the results and send them back to the client browser. To merge the records, they must be collected at a central host, which
obviously is the Tomcat Server as it is the one that has to and cab possibly talk to the client browser. So, the split results are collected by the web-server and merged to form a single text file of results. This text file is then sent to the client browser that is waiting for the results. Figure 3.8 shows the merger method in action.
Figure 3.8: Data Flow Details in Distributed C-Ranker (Merging)
Chapter 4

INFRASTRUCTURE SETUP AND EXECUTION

4.1 Overview

In this chapter, we talk about how C-Ranker is executed and how we obtain the results. We talk about the steps to install and set up the environment for the execution of the program. Moreover, we talk about the improvements that our distributed framework brings out compared to the original single-threaded C-Ranker algorithm.

4.2 Infrastructure Setup

So, before starting the execution of C-Ranker, we need to set up the platform that would execute the programs. To achieve the same, we follow the following steps:

• First we need to install the web server—Apache Tomcat Webserver. In order to so, we navigate to http://tomcat.apache.org/download-70.cgi and download the latest version of Apache Tomcat. Figure 4.1 shows the download links. To see these links scroll down to the section describing binary distributions. Apache Tomcat Binary images are available for both Windows x86 and x64.

• Once we have downloaded the zipped file containing, we extract it in the C: drive. The file is named similar to apache-7.0.37. Figure 4.2 shows the tomcat folder after it is extracted.

• Before starting the Tomcat Server, we setup a few Environment Variables which tell
Figure 4.1: Download Apache Tomcat

Java applications where the Runtime Environment and the Development Kit files are located in the Windows file-system.

To set JAVA_HOME to the directory containing the Java SDK [Software Development Kit], follow the following steps:

– Right Click on My Computer and select Properties.

– In the properties section, there is an option on the left panel named Advanced. In this section, there is the facility for defining new environment variables.

– Add a new variable. Name it JAVA_HOME and set its value to the directory that contains the Java SDK.

– To avoid problems with orphaned DLL files across various windows version, add %JAVA_HOME%/jre/bin/ to the PATH variable. The PATH variable contains a list of folders where Windows would search for executable files.

• Now that we have setup the Java Runtime and the associated variables, it’s time to start the Tomcat Server. To start the server, navigate to the bin directory located
in the `apache-tomcat-version` directory that you extracted in the `C:\` drive.

Figure 4.3 show the binary file to double-click for starting the server. The file is named `startup.bat`.

• Now the server should be up and running. Figure 4.4 shows a screenshot showing a running instance of the server.
Now that the tomcat server has been set up for running, it is time to set up C-Ranker.

We follow the following steps to configure and setup C-Ranker:

• Move the file ROOT.war from the project directory to
  
  C:\apache-tomcat-7.0.62-windows-x64\webapps. Figure 4.5 shows the copied file.

Now it’s time to configure C-Ranker so that it knows the IP addresses of the worker computers in the systems and their respective ports. We also make directories for storing temporary data and for holding the configuration properties.

We follow the following steps in order to accomplish the above mentioned:

• First we make a new folder in C:\ named cranker-temp.

• cranker-read, cranker-solve and cranker-write are stored in the win32 folder. We move this folder to C:\.
• And ServerRunner.jar to C:\Users\Home. ServerRunner.jar is the worker C-Ranker application that we start on each server. Figure 4.6 shows the moved ServerRunner.jar file in the home directory of a user.

• Create a folder named cranker-props and move the file cranker.properties into this directory. cranker.properties contains the information about
the IP addresses of the servers on which we have to run the worker C-Ranker applications. Figure 4.7 shows a sample `cranker.properties`.

![Cranker Properties File](image)

**Figure 4.7: Cranker Properties File**

The first line contains the list of servers. Servers are separated by commas and written in the format `server_ip:port`.

The second line contains the directory that holds the cranker executables, i.e. the directory that we rightfully call `cranker.home`. We set it to `C:\win32`.

The third line contains the directory that holds the temporary files during the course of the execution of the program. We have set it to `C:\cranker-temp`.

### 4.3 Executing C-Ranker

Now we start the application. To do so, we follow the following steps in order:

- Open up a command prompt by searching for `cmd.exe`.
- Now start the server with the following command:

```
java -jar ServerRunner.jar
```
• The application would ask for a port number. Make sure that the port number is the same as the one entered in the properties file. Figure 4.8 shows the ServerRunner.jar running and asking for a port from the user.

Figure 4.8: ServerRunner.jar Running and Asking for A Port Number

• Follow the same procedure listed above for all the servers listed in the cranker.properties file.

• Now, open up the file submitting interface in a browser by visiting http://${ip}:${port} of the tomcat web server.

• A web page with a form for submitting the input file appears. Select a input file from your computer and then hit submit. Figure 4.9 shows the interface for submitting an input file to the system.

• The input file has been submitted and is then run in the distributed system and results made available.
Figure 4.9: Web Based Interface for Submitting Input File
Chapter 5

RESULTS

In this section, we discuss how we obtain the results. We also analyze and discuss the consequences of the distributed approach in contrast to the centralized approach.

We compare the execution times of the two approaches for three different input files of varying size.

The table 5.1 shows the data sets along with their sizes.

The results are obtained by using four servers with the configurations stated below.

These diagrams shows the times for the single server and four server C-Ranker for testData.xls. The single server takes 1 minute 30 seconds while the distributed 4-server C-Ranker takes only 1 minutes, i.e. a single server takes 50% more time than the 4-server approach.

These are the results for the dataset Pbmc_orbit_mips.xls. A single server takes 7 hours 12 minutes and 24 seconds while four servers together take 3 hours 56 minutes and 24 seconds. So, four servers takes almost half the time as compared to the single server.

<table>
<thead>
<tr>
<th>Data (Filename)</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>testData.xls</td>
<td>409KB</td>
</tr>
<tr>
<td>Pbmc_orbit_mips.txt</td>
<td>11,221KB</td>
</tr>
<tr>
<td>Pbmc_orbit_nomips.txt</td>
<td>12,816KB</td>
</tr>
<tr>
<td>Pbmc_velos_mips.txt</td>
<td>31,422KB</td>
</tr>
<tr>
<td>Pbmc_velos_nomips.txt</td>
<td>31,422KB</td>
</tr>
</tbody>
</table>

Table 5.1: Input Data Sets for Observations
<table>
<thead>
<tr>
<th>Servers</th>
<th>Server_1</th>
<th>Server_2</th>
<th>Server_3</th>
<th>Server_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>8GB</td>
<td>4GB</td>
<td>4GB</td>
<td>4GB</td>
</tr>
<tr>
<td>Processor</td>
<td>i5</td>
<td>i5</td>
<td>i5</td>
<td>i5</td>
</tr>
<tr>
<td>OS</td>
<td>Windows 7</td>
<td>Windows Vista</td>
<td>Windows 7</td>
<td>Windows 7</td>
</tr>
</tbody>
</table>

Table 5.2: Hardware Used to Observe Results

Figure 5.1: Results for testData.xls

Figure 5.2: Results for Pbmc_orbit_mips.xls
Figure 5.3: Results for Pbmc_orbit_nomips.xls

These are the results for Pbmc_orbit_nomips.xls dataset. The single server took 12 hours, 24 minutes and 0 seconds while four servers took 8 hours 1 minute and 32 seconds.

Figure 5.4: Results for Pbmc_velos_mips.xls

These are the results for Pbmc_velos_mips.xls dataset. The single server in this
case took 12 hours, 58 minutes and 2 seconds while four servers together took 8 hours, 25 minutes and 43 seconds.

Figure 5.5: Results for Pbmc_velos_nomips.xls

These are the results for Pbmc_velos_nomips.xls. The single server in this case took 14 hours, 2 minutes and 32 seconds while four servers together took 9 hours, 19 minutes and 12 seconds.

The above results show that the execution time taken on four servers to that on a single server is about $\frac{2}{3}$.

5.1 Analysis of Results

5.1.1 Memory Usage

Now, we discuss the memory requirements of the two approaches. It turns out that the distributed approach requires less memory on average as compared to the centralized approach. Figure 5.6 shows the memory usage of the two approaches for increasing size of the input data sets.
Figures 5.7 to 5.11 show the memory usage of the C-Ranker in the centralized vs distributed approach. The chart at the top is for a single server running C-Ranker. The charts below are for 1, 2, 3 and 4 servers used for running C-Ranker. We see a drastic change in memory usage when we use the distributed approach.

These charts show the memory usage of C-Ranker for the testData.xls file. The size
of the data set is 409KB. On a single server, C-Ranker takes 57.4% of the total system memory. The memory usage reduces down to a minimum of 51.6% with three or four servers.

Figure 5.8: Memory Usage

These carts show the memory usage of C-Ranker for Pbmc_orbit_mips.xls. The size of this data set is 11,221KB. On a single server, C-Ranker takes 70.5% of the total system memory while with multiple server, we get an minimum of 54.5% memory usage when we use three servers.

Figure 5.9: Memory Usage
These charts are for Pbmc_orbit_nomips.txt, whose size is 12,816KB. On a single server, it takes 75.2% of the system memory while with the distributed approach, we can get to as low as 53.3%.

Figure 5.10: Memory Usage

These charts are for the dataset: Pbmc_velos_mips.txt. The size of this dataset is 31,422KB. On a single server, single-threaded C-Ranker takes 86.5% of the system memory while with the distributed approach, it takes a minimum of 57.1% memory usage when we have four servers.

Figure 5.11: Memory Usage
These charts are for the dataset: Pbmcl_velos_nomips.txt. The size of this dataset is 48,486KB. On a single server running the original C-Ranker, it takes 89.3% of the system memory to find the results. While with the distributed setting, it takes a minimum of 54.9% memory usage with four servers.

Now, we discuss the memory requirements of the two approaches qualitatively. It turns out that the distributed approach requires less memory on average as compared to the centralized approach. Figure 5.12 shows the memory usage of the two approaches for increasing size of the input data sets.

![Figure 5.12: Memory Usage](image)

Moreover, the difference in the memory usage increases with increase in the size of the input data. Figure 5.13 shows this very fact.

### 5.2 Comparison with C-Ranker in an Apache Hadoop Framework

The Apache Hadoop data processing software is immersed in a complex environment composed of massive machine clusters, large at sets, and several processing jobs.
The framework uses Apache Hadoop Distributed File System (HDFS) and Apache Mapreduce to store and process the peptide data respectively. The framework has two steps: Execute the C-Ranker algorithm on Hadoop cluster and compare the correct PSMs data generated via Hadoop approach with the normal execution approach of C-Ranker [Donepudi, 2015]. The number of servers in The Apache Hadoop framework depends on the block size specified and the input data size. For an input of data size 11221 KB and a block size of 250, the input will be divided into 45 servers. As Amazon EC2 micro instances are used in Apache Hadoop’s framework, using only 4 servers is not feasible. Similarly, Using 45 servers for our framework is not feasible as we need to have 45 commodity hardware.

The comparison of our results with the Apache Hadoop’s results are shown in Table 5.3.

Initially, our results were quite bad when compared to clustered 2 Hadoop. So, we had to upgrade the hardware to check the performance for our framework. Our updated hardware is as shown in Table 5.4 and comparison with cluster2 Hadoop is as shown in Table 5.5.
<table>
<thead>
<tr>
<th>PBMC data (KB)</th>
<th>C-Ranker Execution time in hrs (Cluster 1 Hadoop)</th>
<th>Distributed approach for C-Ranker execution time in hrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>11221</td>
<td>6.5</td>
<td>3.56</td>
</tr>
<tr>
<td>12816</td>
<td>9.9</td>
<td>8.1</td>
</tr>
<tr>
<td>31422</td>
<td>10.2</td>
<td>8.25</td>
</tr>
<tr>
<td>48486</td>
<td>15.2</td>
<td>9.2</td>
</tr>
</tbody>
</table>

Table 5.3: Comparison of C-Ranker on distributed approach with C-Ranker on an Apache Hadoop Framework Cluster 1

<table>
<thead>
<tr>
<th>Servers</th>
<th>Server_1</th>
<th>Server_2</th>
<th>Server_3</th>
<th>Server_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>12GB</td>
<td>8GB</td>
<td>12GB</td>
<td>8GB</td>
</tr>
<tr>
<td>Processor</td>
<td>i7</td>
<td>i5</td>
<td>i7</td>
<td>i7</td>
</tr>
<tr>
<td>Operating System</td>
<td>Windows 8</td>
<td>Windows 7</td>
<td>Windows 7</td>
<td>Windows 7</td>
</tr>
</tbody>
</table>

Table 5.4: Upgraded hardware to compare with cluster 2 Hadoop

<table>
<thead>
<tr>
<th>PBMC data (KB)</th>
<th>Distributed approach for C-Ranker execution time in hrs (new results)</th>
<th>C-Ranker Execution time in hrs (Cluster 2 Hadoop)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11221</td>
<td>1.7</td>
<td>1.3</td>
</tr>
<tr>
<td>12816</td>
<td>3.82</td>
<td>1.58</td>
</tr>
<tr>
<td>31422</td>
<td>4.1</td>
<td>3.4</td>
</tr>
<tr>
<td>48486</td>
<td>5.93</td>
<td>4.5</td>
</tr>
</tbody>
</table>

Table 5.5: Comparison of C-Ranker on distributed approach with C-Ranker on an Apache Hadoop Framework Cluster 2
Table 5.6: Cost Calculation of Apache Hadoop Cluster 1 and Cluster 2

<table>
<thead>
<tr>
<th>PBMC Data Size(KB)</th>
<th>Cost of Hadoop Cluster 1($)</th>
<th>Cost of Hadoop Cluster 2($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11221</td>
<td>3.4581</td>
<td>0.6916</td>
</tr>
<tr>
<td>12816</td>
<td>5.2668</td>
<td>0.8410</td>
</tr>
<tr>
<td>31422</td>
<td>5.4264</td>
<td>1.8088</td>
</tr>
<tr>
<td>48486</td>
<td>8.8064</td>
<td>2.3941</td>
</tr>
</tbody>
</table>

While big data isn’t solely made for huge organizations, not all huge information stages are suited for little information needs. Sadly, Hadoop happens to be one of them. Because of its high capacity design, the Hadoop Distributed File System or HDFS, does not have the capacity to effectively bolster the arbitrary reading of little records. Subsequently, it is not suggested for associations with small amounts of information [Dong, Qiu, Zheng, Zhong, Li, and Li, 2010]. The MapReduce framework is extremely difficult to leverage for more than our simple transformational logic.

In Apache Hadoop’s framework, the user specifies the block size and the HDFS automatically divides into number of servers where C-Ranker is processed. Whereas, in our approach, user himself/herself has the privilege to specify the number of servers to be divided. For small files, number of servers to be divided in HDFS framework are high which degrades the performance.

The Apache Hadop’s framework uses Amazon EC2 instances to run the C-Ranker and these instances are not free. They cost $0.532 per hour. Our framework is absolutely free.
Chapter 6

CONCLUSION AND FUTURE WORK

Implementation of C-Ranker algorithm has been done in a distributed setting and ended up with great results. The nature of the solution calls for parallel execution and hence a decision to implement the same in Java has been taken. The results clearly show that distributed C-Ranker executes in less time as compared to the conventional centralized C-Ranker application. It is observed that, around half overall reduction in execution time is shown with the addition of the second server. The execution time does decrease with the addition of more and more servers, but this increase is lower, just like the law of diminishing returns. But in this case, it is the overhead introduced with managing and coordinating the different servers that increases with the number of servers added to the system. So an important decision while implementing such a system is to find out the number servers that we need to deploy. It is still an open research problem as to how one can automatically chooses this number. Our framework was tested with 1, 2, 3 and 4 servers and saw the results for each. Moreover, there is reduction in the average memory usage with the distributed system running C-Ranker on multiple servers. An important benefit that may get overlooked is the fact the distributed C-Ranker can be used to solve extraordinarily large problems without incurring expenses for a powerful computer or super computer. It is more expensive to buy a single high-performance computer rather than buying a few cheaper commodity hardware computers and using them as a cluster. With distributed C-
Ranker, solutions to larger peptide identification problems become tractable at a lower cost as compared to achieving the same in a centralized setting.

Distributed C-Ranker is scalable. Scaling the algorithm simply requires adding an extra computer running the C-Ranker server application and updating the cranker.properties file to include the ip addresses of the new servers, along with their respective ports. There are improvements that can be made in the application. It can be made to have a better interface for installation. Moreover, the copying of files can be automated with batch files so that someone installing the application should not worry about the details. A guiding GUI may be helpful too. The same distributed approach can be used with Percolator and PeptideProphet to see how well they perform. Additionally, once can use an ensemble method to combine the results of the three tools. Majority vote can be taken on the results returned by the three tools, meaning we may mark a PSM reliable only if at least two of the methods declare it reliable. A web interface can also be implemented to have a dynamic configuration, such as a method for the user to change the number of servers through its interface. It would make the scaling effortless for users.


Mantena, S. R. Transparency in distributed systems.


Appendices
Appendix A
SOURCE CODE COMMENTARY
A.A The calcExcel Method

This method, as is evident below, is reading the response of the from worker servers
and storing the result in `r.file` which is an attribute of the `Result` object. This method
also validates the response obtained and raises an exception if the response received is
malformed.

```java
package ru.ingie.cranker.client;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

public class CRanker {

    static class Result {
        File file;
        Exception ex;
    }

    public File calcExcel(String sourcePath, String servers) throws
            IOException, ExecutionException, InterruptedException {
        String[] servs = servers.split(".
        List<File> parts = new ExcelDivider().divide(sourcePath, servs.
            length);
        for (int i = 0; i < parts.size(); i++) {
            System.out.println(parts.length+"::”+parts.get(i).
                getAbsolutePath());
        }
        ExecutorService es = Executors.newFixedThreadPool(servers.length);
        List<Future<Result>> futures = new ArrayList<Future<Result>>()
        for (int i = 0; i < parts.size(); i++) {
            final CRankerClient client = new CRankerClient(servers[i].
                split(".")[0], Integer.parseInt(servers[i].split(":
                    ")[1]));
            final File file = parts.get(i);
            Future<Result> f = es.submit(new Callable<Result>() {
                @Override
                public Result call() {
                    Result r = new Result();
                    try {
                        r.file = client.calcExcel(file);
                    } catch (Exception ex) {
                        r.ex = ex;
                    }
                }
            });
```
public File calcTxt(String sourcePath, String servers) throws IOException, ExecutionException, InterruptedException {
    String[] servs = servers.split(" ");
    List<File> parts = new TxtDivider().divide(sourcePath, servs.length);
    ExecutorService es = Executors.newFixedThreadPool(servs.length);
    List<Future<Result>> futures = new ArrayList<Future<Result>>();
    for (int i = 0; i < parts.size(); i++) {
        final CRankerClient client = new CRankerClient(servs[i].split(":")[0], Integer.parseInt(servs[i].split(":")[1]));
        final File file = parts.get(i);
        Future<Result> f = es.submit(new Callable<Result>() {
            @Override
            public Result call() {
                Result r = new Result();
                try {
                    r.file = client.calcTxt(file);
                } catch (Exception ex) {
                    r.ex = ex;
                }
                return r;
            }
        });
        futures.add(f);
    }
    List<File> files = new ArrayList<File>();
    for (Future<Result> f : futures) {
        Result r = f.get();
        if (r.ex != null) {
            throw new RuntimeException("One of worker failed", r.ex);
        } else {
            //add file to array
            files.add(r.file);
        }
    }
    System.out.println("All files received. Merging them");
    File mergeResults = new TextToExcelMerger().merge(files);
    System.out.println("Merged to:" + mergeResults.getAbsolutePath());
    return mergeResults;
}
} else {
    // add file to array
    files.add(r.file);
}

System.out.println("All files received. Merging them");
File mergeResults = new TextToTextMerger().merge(files);
System.out.println("Mergeed to:" + mergeResults.getAbsolutePath());

return mergeResults;
}
A.B Excel Divider driver

There are two divide methods, one defined in the ExcelDivider class and similar one defined in the TextDivider class. As the name suggests, the divide methods is a method that splits the input file into multiple files equal to the number of servers available for computation. In the source code below, the signature of the function takes as input `path` to the input .xls file and an `int` namely `parts` which is the number of pieces into which the input .xls file would be divided. The function returns a list of `File` objects, i.e. `List<File>`.

```java
package ru.ingie.cranker.client;

import org.apache.poi.hssf.usermodel.*;
import org.apache.poi.ss.usermodel.Cell;
import org.apache.poi.ss.usermodel.CellStyle;

import java.io.*;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

public class ExcelDivider {

    public static void main(String[] args) throws IOException {
        if (args.length < 2) {
            usage();
        }
        ExcelDivider process = new ExcelDivider();
        process.divide(args[0], Integer.parseInt(args[1]));
    }

    private static void usage() {
        System.out.println("Usage: ExcelDivider [path] [parts] ");
        System.exit(1);
    }

    public List<File> divide(String path, int parts) throws IOException {
        List<File> result = new ArrayList<File>();
        if (parts <= 0) {
            throw new IllegalArgumentException("Number of parts must be a positive number");
        }
        FileInputStream fis = null;
        try {
            int firstDataRow = 0;
            List<String> columns = new ArrayList<String>();
            fis = new FileInputStream(path);
            HSSFWorkbook wb = new HSSFWorkbook(fis);
            HSSFSheet sh = wb.getSheetAt(0);
            for (int i = sh.getFirstRowNum(); i <= sh.getLastRowNum(); i += 1) {
                HSSFRow row = sh.getRow(i);
                if (row.getLastCellNum() == 0) {
                    columns.add(row.getCell(0).getStringCellValue());
                }
            }
        } finally {
            fis.close();
        }
```
continue;
}
HSSFCell cell = row.getCell(0);
if (cell.getStringCellValue() == null || cell.
getStringCellValue().equals("")) {
    continue;
}
System.out.println("Starting from row "+ i);
System.out.println("Detecting columns...");
for (int j = row.getFirstCellNum(); j <= row.\ngetLastCellNum(); j++) {
    cell = row.getCell(j);
    if (cell == null) {
        break;
    }
    String columnName = cell.getStringCellValue();
    System.out.println("Detected: "+ columnName);
    columns.add(columnName);
}
firstDataRow = i + 1;
b
break;
}
int dataRows = sh.getLastRowNum() - firstDataRow + 1;
if (dataRows < parts) {
    System.out.println("Number of parts truncated to "+
dataRows);
    parts = dataRows;
}
System.out.println("Dividing App. rows in part will be " +
(dataRows / parts));
int lastToProcess = firstDataRow - 1;
for (int i = 0; i < parts; i++) {
    File file = File.createTempFile("crancker", ".xls");
    int firstToProcess = lastToProcess + 1;
    if (i == parts - 1) {
        lastToProcess = sh.getLastRowNum();
    }
    else {
        lastToProcess = firstToProcess + (dataRows / parts) - 1;
    }
    fillFile(file, sh, firstToProcess, lastToProcess,
    columns);
    result.add(file);
    System.out.println("Part "+ (i +1) + ",("+
    firstToProcess + ",;"+ lastToProcess + ") prepared. See "+ file.
    getAbsolutePath());
}
wb.close();
}
finally {
    if (fis != null) {
        try {
            fis.close();
        }
    }
}
```java
} catch (IOException e) {
    // silent
}

private void fillFile(File file, HSSFSheet original, int firstToProcess, int lastToProcess, List<String> columns) throws IOException {
    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sh = wb.createSheet("sheet1");
    int i = 0;
    HSSFRow row = sh.createRow(i++);
    HashMap<Integer, CellStyle> columnStyles = new HashMap<Integer, CellStyle>();
    for (int j = 0; j < columns.size(); j++) {
        HSSFCell cell = row.createCell(j);
        cell.setCellValue(columns.get(j));
    }
    for (int oi = firstToProcess; oi <= lastToProcess; oi++) {
        HSSFRow sourceRow = original.getRow(oi);
        HSSFRow newRow = sh.createRow(i++);
        for (int j = 0; j < columns.size(); j++) {
            // Grab a copy of the old/new cell
            HSSFCell oldCell = sourceRow.getCell(j);
            HSSFCell newCell = newRow.createCell(j);

            // If the old cell is null jump to next cell
            if (oldCell == null) {
                newCell = null;
                continue;
            }

            // Set the cell data type
            newCell.setCellType(oldCell.getCellType());
            if (!columnStyles.containsKey(j)) {
                CellStyle columnStyle = wb.createCellStyle();
                columnStyle.setFont(oldCell.getCellStyle().getFont(original.getWorkbook()));
                columnStyle.setDataFormat(oldCell.getCellStyle().getDataFormat());
                columnStyles.put(j, columnStyle);
            }
            newCell.setCellValue(oldCell.getStringCellValue());
            // Set the cell data value
            switch (oldCell.getCellType()) {
                case Cell.CELL_TYPE_BLANK:
                    break;
                case Cell.CELL_TYPE_BOOLEAN:
```
newCell.setValue(oldCell.getBooleanCellValue());
            break;
        case Cell.CELL_TYPE_ERROR:
            newCell.setValue(oldCell.getErrorCellValue());
            break;
        case Cell.CELL_TYPE_FORMULA:
            newCell.setValue(oldCell.getCellFormula());
            break;
        case Cell.CELL_TYPE_NUMERIC:
            if (oldCell.getStyle().getDataFormatString().contains("/")){
                newCell.setValue(oldCell.getDateCellValue());
            } else {
                newCell.setValue(oldCell.getNumericCellValue());
            }
            break;
        case Cell.CELL_TYPE_STRING:
            // System.out.println("Value::"+oldCell.getStringCellValue());
            newCell.setValue(oldCell.getStringCellValue());
            break;
        }
    }
    wb.write(new FileOutputStream(file));
    wb.close();
}

So, to summarize, the functions performed by the **divide** method are:

- Path of the input file and number of parts are inputs for this method. For example: `C:\Users\...\testData.xls` is the path and 4 is the parts.
- If the file size is 100KB, we get four 25KB files.
- Output is four partial excel files.
A.C The merge Method

The `merge` method takes a `List` of `File` objects and merges them together by concatenating the rows into a single file. There are two variants of the merge methods that we use, one defined in `TextToExcelMerger` class and the other in `TextToTextMerger` class. We need two such methods because the C-Ranker tools results are text files while the inputs are .xls files in our case.

```java
package ru.ingie.cranker.client;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;

import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.ss.usermodel.Cell;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.ss.usermodel.Workbook;

public class TextToExcelMerger {

    /**
     * Merges a list of text files into 1 excel file
     * @param files
     * @return
     * @throws IOException
     */
    public File merge(List<File> files) throws IOException {
        // create the main document
        HSSFWorkbook wb = new HSSFWorkbook();

        // add the sheet
        Sheet sh = wb.createSheet("Sheet1");
        String[] columnNames = null;
        int currentRow = 0;

        // we go through the files appending their results
        for (File file: files)
            // we go through the file until we find the header row
            BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(file)));
            String line = reader.readLine();
            while (line != null) {
                String[] splits = line.split("\t");
                if (splits.length > 4) {
                    // we have found the header
                    if (columnNames == null) {
                        columnNames = splits;
                    } else {
                        Row firstRow = sh.createRow(currentRow);
                        for (int i = 0; i < splits.length; i++) {
                            firstRow.createCell(i).setCellValue(splits[i]);
                        }
                    }
                }
                currentRow++;
            }

        // write the document
        try {
            FileOutputStream fos = new FileOutputStream("output.xlsx");
            wb.write(fos);
            fos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

        return null;
    }
}
```
currentRow++;

for (int index=0; index<columnNames.length; index++) {
    Cell cell = firstRow.createCell(index);
    cell.setCellValue(columnNames[index]);
}

break;

line = reader.readLine();
}

// now we have reached where data is
line = reader.readLine();
while (line != null) {
    String[] splits = line.split("\t");
    if (splits.length == columnNames.length) {
        // we add the row
        Row newRow = sh.createRow(currentRow);
        currentRow++;
        for (int index=0; index<splits.length; index++) {
            Cell cell = newRow.createCell(index);
            cell.setCellValue(splits[index]);
        }
        line = reader.readLine();
    }
    reader.close();
}

// we have added all the files we now create a disk excel file
File resFile = File.createTempFile("mergeres", "");
FileOutputStream out = new FileOutputStream(resFile);
wb.write(out);
wb.close();
out.close();

return resFile;
A.D The serve Method

The serve method is a part of the CRankerServer class, whose instance objects are created on each node of the distributed network. This class maintains the state of each worker host. The serve method is the only instance method of this class. It is responsible for running the cranker commands in sequence and generating the intermediate files and the slit output file.

So, to summarize the server methods does the following:

- Waits for connection on specified port. If exists, accepts the connection
- Read the cranker home dir from home system. A random id is generated and saved for partial excel files.
- Cranker-read process is executed. At this point, we have first matlab file.
- Similarly, cranker-solve and cranker-write processes are executed.
- Files will be written back in sequence and generating the intermediate files and the slit output file

```
package ru.ingie.cranker.server;

import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.ByteBuffer;
import java.nio.file.Files;
import java.util.Properties;
import java.util.UUID;

/**
 * @author ingie
 * @since 16.02.2015
 */
public class CRankerServer {
    private ServerSocket servsock = null;

    public void serve(int port) throws IOException {
        try {
            servsock = new ServerSocket(port);
            while (true) {
                System.out.println("Waiting for connection on port " + port);
                try {
                    final Socket sock = servsock.accept();
                    System.out.println("Accepted connection : " + sock);
                    new Thread(new Runnable() {
                        @Override
                        public void run() {
                            FileInputStream fis = null;
                            BufferedOutputStream bos = null;
                            InputStream is = null;
                            OutputStream os = null;
                            int current, read;
                            try {
                                try {
                                    // Code...
                                }
                                catch (Exception e) {
                                    // Code...
                                }
                            }
                        }
                    });
                }
            }
```

67
is = sock.getInputStream();
byte[] arr = new byte[4];
is.read(arr, 0, 4);
ByteBuffer wrapped = ByteBuffer.wrap(arr);
int fileSize = wrapped.getInt();

System.out.println("File size will be "+fileSize+" bytes");
// read the cranker home dir from file system
Properties props = new Properties();
props.load(new FileInputStream(
    new File(System
        .getProperty("user.home")
        + File.separator
        + "cranker−props"+
        + File.separator
        + "cranker.properties")));

byte[] bytes = new byte[fileSize];
File temp = new File(new File(props.getProperty("temp.
dir")), UUID.randomUUID().toString()+".xls");
fos = new FileOutputStream(temp);
bos = new BufferedOutputStream(fos);

current = 0;
do {
    read = is.read(bytes, current, (bytes.length−current));
    current += read;
} while (read > 0);

bos.write(bytes, 0, current);
bos.flush();
System.out.println("File received: "+
    temp.getAbsolutePath());
System.out.println("Running cranker on the file");
final UUID uuid = UUID.randomUUID();
File tempRes1 = new File(new File(props.getProperty("temp.
dir")), uuid.toString()+".mat");
String command = "\""+props.getProperty("cranker.home")+
    "cranker_read.exe\""+
    "\""+temp.getAbsolutePath() + "\" \"
    + tempRes1.getAbsolutePath() + "\" ";
System.out.println("Command::"+command);
ProcessBuilder builder = new ProcessBuilder(
    command);
builder.directory(new File(props
    .getProperty("cranker.home")));
Process p=builder.start();
StringBuffer output = new StringBuffer();
p.waitFor();
BufferedReader reader =
    new BufferedReader(new
    InputStreamReader(p.getInputStream()));
String line = "";
while ((line = reader.readLine()) != null) {
    output.append(line + "\n");
}
System.out.println("Cranker Response\n" + output);
// at this point we have the first matlab file.
// we execute the second command
final UUID uuid2 = UUID.randomUUID();
File tempRes2 = new File(new File(props.getProperty("temp.dir")), uuid2.toString() + ".mat");
String command2 = "\"" + props.getProperty("cranker.home") + "cranker_solve.exe\"" + 
    "\"" + tempRes1.getAbsolutePath() + "\" " 
    + tempRes2.getAbsolutePath() + "\" ";
System.out.println("Command2:\"" + command2);
bUILDER = new ProcessBuilder(command2);
bUILDER.directory(new File(props.getProperty("cranker.home")));
p = BUILDER.start();
output = new StringBuffer();
p.waitFor();
reader = new BufferedReader(new InputStreamReader(p.getInputStream()));
line = "";
while ((line = reader.readLine()) != null) {
    output.append(line + "\n");
}
System.out.println("Cranker Response 2:\n" + output);

// at this point we have the second matlab
// results file. Now we generate the third file
String command3 = "\"" + props.getProperty("cranker.home") + "cranker_write.exe\"" + 
    "\"" + tempRes1.getAbsolutePath() + "\" " 
    + tempRes2.getAbsolutePath() + "\" ";
System.out.println("Command3:\"" + command3);
bUILDER = new ProcessBuilder(command3);
bUILDER.directory(new File(props.getProperty("cranker.home")));
p = BUILDER.start();
output = new StringBuffer();
p.waitFor();
reader = new BufferedReader(new InputStreamReader(p.getInputStream()));
line = "";
while ((line = reader.readLine()) != null) {
    output.append(line + "\n");
}
System.out.println("Cranker Response 3:\n" + output);

// we now search for the generated file
temp.delete();
tempRes1.delete();
tempRes2.delete();
System.out.println("Look for file ::"+uuid.toString());
File crankerTemp = new File(props
.getProperty("temp.dir"));
String[] files = crankerTemp
.list(new FilenameFilter() {
    @Override
    public boolean accept(File arg0,
        String name) {
        return name.startsWith(uuid
            .toString())&& name.toLowerCase().contains(".txt");
    }
});
// if we have more than one file matching then
// there was an error
System.out.println("Files found ::"+files.length);
if (files.length == 1) {
    File theFile = new File(crankerTemp,
        files[0]);
    System.out.println("Output file found ::"+theFile.
        getAbsolutePath());
    // we write that whole file to the client
    int size = (int) Files.size(theFile
        .toPath());
    ByteBuffer buf = ByteBuffer.allocate(4);
    buf.putInt(size);
    os = sock.getOutputStream();
    os.write(buf.array());
    os.flush();
    byte[] bytes = new byte[size];
    FileInputStream fis = new FileInputStream(
        theFile);
    BufferedReader bis = new BufferedReader(
        new InputStreamReader(
            fis));
    int read = bis.read(bytes, 0, bytes.length);
    if (read != bytes.length) {
        bis.close();
        throw new RuntimeException(
            "Can't load file (size differs)");
    }
    os.write(bytes, 0, bytes.length);
    os.flush();
    System.out
        .println("Written back results file");
    // we delete all the files from cranker hime
    theFile.delete();
    bis.close();
}
} catch (IOException e) {
    e.printStackTrace();
} catch (InterruptedException e) {
e.printStackTrace();
}
finally {
  if (bos != null)
    try {
      bos.close();
    }
    catch (IOException e) {
    }
  if (fos != null)
    try {
      fos.close();
    }
    catch (IOException e) {
    }
  if (os != null)
    try {
      os.close();
    }
    catch (IOException e) {
    }
  if (is != null)
    try {
      is.close();
    }
    catch (IOException e) {
    }
  if (sock != null)
    try {
      sock.close();
    }
    catch (IOException e) {
    }
  }}
}).start();
}
finally {
}
}
finally {
  if (servsock != null)
    servsock.close();
}